

WEEK 1: Practice Problems (1–10): Complete solutions + Java reference implementations

Problem 1: Username Availability Checker

Requirements: O(1) check, suggestions, track attempted popularity.

Design

- HashSet<String> taken or HashMap<String, Integer> usernameToUserId
- HashMap<String, Integer> attempts for popularity
- Suggest: append numbers, replace _ with ., etc.

```
import java.util.*;  
  
public class UsernameChecker {  
    private final Set<String> taken = new HashSet<>();  
    private final Map<String, Integer> attempts = new HashMap<>();  
    public UsernameChecker(Collection<String> existing) {  
        taken.addAll(existing);  
    }  
    public boolean checkAvailability(String username) {  
        attempts.put(username, attempts.getOrDefault(username, 0) + 1);  
        return !taken.contains(username);  
    }  
}
```

```
}

public List<String> suggestAlternatives(String username, int limit) {

    List<String> out = new ArrayList<>();

    // simple deterministic suggestions

    for (int i = 1; out.size() < limit && i <= 50; i++) {

        String c = username + i;

        if (!taken.contains(c)) out.add(c);

    }

    if (out.size() < limit) {

        String dot = username.replace('_', '!');

        if (!taken.contains(dot)) out.add(dot);

    }

    return out;

}

public String getMostAttempted() {

    String best = null;

    int bestCnt = -1;

    for (var e : attempts.entrySet()) {

        if (e.getValue() > bestCnt) {

            bestCnt = e.getValue();

            best = e.getKey();

        }

    }

    return best;

}
```

Problem 2: Flash Sale Inventory Manager (no overselling)

Hints mention HashMap + concurrency + waiting list.

Design

- ConcurrentHashMap<String, AtomicInteger> stock
- ConcurrentHashMap<String, ArrayDeque<Long>> waitingList (synchronized per product)

```
import java.util.*;  
  
import java.util.concurrent.*;  
  
import java.util.concurrent.atomic.*;  
  
public class FlashSaleInventory {  
  
    private final ConcurrentHashMap<String, AtomicInteger> stock = new  
    ConcurrentHashMap<>();  
  
    private final ConcurrentHashMap<String, ArrayDeque<Long>> waiting = new  
    ConcurrentHashMap<>();  
  
    public void addProduct(String productId, int initialStock) {  
  
        stock.put(productId, new AtomicInteger(initialStock));  
  
        waiting.put(productId, new ArrayDeque<>());  
  
    }  
  
    public int checkStock(String productId) {  
  
        return stock.get(productId).get();  
  
    }  
  
    public String purchaseItem(String productId, long userId) {  
  
        AtomicInteger cnt = stock.get(productId);  
  
        while (true) {  
  
            int cur = cnt.get();  
  
            if (cur <= 0) {  
  
            }  
  
        }  
  
    }  
}
```

```

        synchronized (waiting.get(productId)) {
            waiting.get(productId).addLast(userId);
            return "WAITLIST position=" + waiting.get(productId).size();
        }
    }

    if (cnt.compareAndSet(cur, cur - 1)) {
        return "SUCCESS remaining=" + (cur - 1);
    }
}

}

```

Problem 3: DNS Cache with TTL + LRU

Your problem requires TTL expiration + stats + LRU.

Design

- HashMap<String, Entry> for O(1)
- LinkedHashMap in access-order for LRU eviction
- Store expiresAtMillis, remove expired on access + periodic cleanup thread.
-

```

import java.util.*;

public class DnsCache {

    static class Entry {
        final String ip;
        final long expiresAt;

        Entry(String ip, long expiresAt) { this.ip = ip; this.expiresAt = expiresAt; }

        boolean expired(long now) { return now >= expiresAt; }
    }
}

```

```
}

private final int capacity;

private long hits = 0, misses = 0;

// accessOrder=true gives LRU behavior

private final LinkedHashMap<String, Entry> cache;

public DnsCache(int capacity) {

    this.capacity = capacity;

    this.cache = new LinkedHashMap<>(16, 0.75f, true) {

        @Override protected boolean removeEldestEntry(Map.Entry<String, Entry> eldest) {

            return size() > DnsCache.this.capacity;

        }

    };

}

public synchronized String resolve(String domain) {

    long now = System.currentTimeMillis();

    Entry e = cache.get(domain);

    if (e != null && !e.expired(now)) {

        hits++;

        return e.ip;

    }

    if (e != null) cache.remove(domain); // expired

    misses++;

    // simulate upstream lookup (replace with real DNS call)

    String ip = "1.2.3.4";

    long ttlMs = 300_000;

    cache.put(domain, new Entry(ip, now + ttlMs));

}
```

```

        return ip;
    }

    public synchronized String stats() {
        long total = hits + misses;

        double hitRate = total == 0 ? 0.0 : (hits * 100.0 / total);

        return "HitRate=" + hitRate + "% hits=" + hits + " misses=" + misses + " size=" +
cache.size();
    }

}

```

Problem 4: Plagiarism Detector with n-grams

Needs n-grams mapped to documents + similarity.

Design

- Build index: `HashMap<String, Set<Integer>> ngramToDocIds`
- For a new doc: count matches per docId → similarity = matches / totalNgrams

```

import java.util.*;

public class PlagiarismDetector {

    private final Map<String, Set<Integer>> index = new HashMap<>();
    private final int n;

    public PlagiarismDetector(int n) { this.n = n; }

    public void addDocument(int docId, String text) {
        for (String ng : ngrams(text, n)) {
            index.computeIfAbsent(ng, k -> new HashSet<>()).add(docId);
        }
    }
}

```

```

public Map<Integer, Double> checkSimilarity(String text) {

    List<String> grams = ngrams(text, n);

    Map<Integer, Integer> matchCount = new HashMap<>();

    for (String g : grams) {

        Set<Integer> docs = index.get(g);

        if (docs == null) continue;

        for (int docId : docs) {

            matchCount.put(docId, matchCount.getOrDefault(docId, 0) + 1);

        }

    }

    Map<Integer, Double> similarity = new HashMap<>();

    for (var e : matchCount.entrySet()) {

        similarity.put(e.getKey(), e.getValue() * 100.0 / grams.size());

    }

    return similarity;

}

private static List<String> ngrams(String text, int n) {

    String[] w = text.toLowerCase().replaceAll("[^a-z\\s]", " ").trim().split("\\s+");

    List<String> out = new ArrayList<>();

    for (int i = 0; i + n - 1 < w.length; i++) {

        StringBuilder sb = new StringBuilder();

        for (int k = 0; k < n; k++) {

            if (k > 0) sb.append(' ');

            sb.append(w[i + k]);

        }

        out.add(sb.toString());

    }

}

```

```

    }
    return out;
}
}

```

Problem 5: Real-time Website Analytics (top pages, uniques, sources)

This is frequency counting + multiple hash maps.

Design

- Map<String, Integer> pageViews
- Map<String, Set<String>> uniqueVisitors
- Map<String, Integer> sourceCounts
- “Top 10” via min-heap or sorting on demand

```

import java.util.*;

public class TrafficAnalytics {

    private final Map<String, Integer> pageViews = new HashMap<>();
    private final Map<String, Set<String>> unique = new HashMap<>();
    private final Map<String, Integer> sources = new HashMap<>();

    public void processEvent(String url, String userId, String source) {
        pageViews.put(url, pageViews.getOrDefault(url, 0) + 1);
        unique.computeIfAbsent(url, k -> new HashSet<>()).add(userId);
        sources.put(source, sources.getOrDefault(source, 0) + 1);
    }

    public List<String> topPages(int k) {
        List<String> urls = new ArrayList<>(pageViews.keySet());

```

```

        urls.sort((a,b) -> Integer.compare(pageViews.get(b), pageViews.get(a)));

        return urls.subList(0, Math.min(k, urls.size()));

    }

}

```

Problem 6: Rate Limiter (token bucket)

Spec calls for HashMap per client + time-based operations.

Design

- ConcurrentHashMap<String, Bucket>
- Bucket: tokens, lastRefillTime
- Refill based on elapsed time, atomic/synchronized per bucket

```

import java.util.concurrent.*;
import java.util.concurrent.atomic.*;

public class TokenBucketLimiter {

    static class Bucket {

        final int maxTokens;

        final double refillPerSec;

        double tokens;

        long lastRefillNanos;

        Bucket(int maxTokens, double refillPerSec) {

            this.maxTokens = maxTokens;

            this.refillPerSec = refillPerSec;

            this.tokens = maxTokens;

            this.lastRefillNanos = System.nanoTime();

        }
    }
}

```

```

}

private final ConcurrentHashMap<String, Bucket> buckets = new
ConcurrentHashMap<>();

public boolean allow(String clientId, int maxPerHour) {

    double refillPerSec = maxPerHour / 3600.0;

    Bucket b = buckets.computeIfAbsent(clientId, k -> new Bucket(maxPerHour,
refillPerSec));

    synchronized (b) {

        long now = System.nanoTime();

        double elapsedSec = (now - b.lastRefillNanos) / 1e9;

        b.tokens = Math.min(b.maxTokens, b.tokens + elapsedSec * b.refillPerSec);

        b.lastRefillNanos = now;

        if (b.tokens >= 1.0) {

            b.tokens -= 1.0;

            return true;
        }
    }

    return false;
}
}

```

Problem 7: Autocomplete (HashMap + Trie hybrid)

Problem explicitly hints Trie + HashMap.

Design

- `HashMap<String, Integer> freq`
- Trie nodes hold top suggestions (store top K queries per prefix)
- Update: increment freq + reinsert into trie path

```
import java.util.*;

public class AutocompleteSystem {

    private final Map<String, Integer> freq = new HashMap<>();

    static class Node {
        Map<Character, Node> child = new HashMap<>();
        List<String> top = new ArrayList<>();
    }

    private final Node root = new Node();

    private static final int K = 10;

    public void updateFrequency(String query) {
        int newFreq = freq.getOrDefault(query, 0) + 1;
        freq.put(query, newFreq);

        Node cur = root;
        for (char c : query.toCharArray()) {
            cur = cur.child.computeIfAbsent(c, x -> new Node());
            updateTopList(cur.top, query);
        }
    }

    public List<String> search(String prefix) {
        Node cur = root;
        for (char c : prefix.toCharArray()) {
            cur = cur.child.get(c);
            if (cur == null) return Collections.emptyList();
        }
        return new ArrayList<>(cur.top);
    }
}
```

```
}

private void updateTopList(List<String> top, String query) {
    if (!top.contains(query)) top.add(query);
    top.sort((a, b) -> {
        int fa = freq.getOrDefault(a, 0);
        int fb = freq.getOrDefault(b, 0);
        if (fa != fb) return Integer.compare(fb, fa); // higher freq first
        return a.compareTo(b); // tie-breaker
    });
    if (top.size() > K) {
        top.subList(K, top.size()).clear();
    }
}

public List<String> searchWithSimpleTypo(String prefix) {
    List<String> direct = search(prefix);
    if (!direct.isEmpty()) return direct;
    if (prefix.length() > 1) {
        return search(prefix.substring(0, prefix.length() - 1));
    }
    return Collections.emptyList();
}

public static void main(String[] args) {
    AutocompleteSystem ac = new AutocompleteSystem();
    ac.updateFrequency("java tutorial");
    ac.updateFrequency("java tutorial");
    ac.updateFrequency("javascript");
}
```

```

        ac.updateFrequency("java download");

        System.out.println(ac.search("jav")); // top suggestions
    }

}

```

Problem 8: Parking Lot with Open Addressing (linear probing)

This is literally in your notes and your problem PDF.

```

import java.util.*;

public class ParkingOpenAddressing {

    enum State { EMPTY, OCCUPIED, DELETED }

    static class Slot {

        String plate;
        State state = State.EMPTY;
        long inTime;

    }

    private final Slot[] table;

    private int size = 0;

    public ParkingOpenAddressing(int capacity) {

        table = new Slot[capacity];
        for (int i = 0; i < capacity; i++) table[i] = new Slot();

    }

    private int hash(String key) {

        return Math.abs(key.hashCode()) % table.length;

    }
}

```

```
public boolean park(String plate) {  
    if (size >= table.length * 0.7) return false; // keep load factor safe for probing  
    int idx = hash(plate);  
    for (int step = 0; step < table.length; step++) {  
        int i = (idx + step) % table.length;  
        if (table[i].state == State.EMPTY || table[i].state == State.DELETED) {  
            table[i].plate = plate;  
            table[i].state = State.OCCUPIED;  
            table[i].inTime = System.currentTimeMillis();  
            size++;  
            return true;  
        }  
    }  
    return false;  
}  
  
public boolean exit(String plate) {  
    int idx = hash(plate);  
    for (int step = 0; step < table.length; step++) {  
        int i = (idx + step) % table.length;  
        if (table[i].state == State.EMPTY) return false;  
        if (table[i].state == State.OCCUPIED && plate.equals(table[i].plate)) {  
            table[i].state = State.DELETED;  
            size--;  
            return true;  
        }  
    }  
}
```

```

        return false;
    }
}

```

Problem 9: Two-Sum variants + duplicates

Problem statement is clear.

```

import java.util.*;

public class TwoSum {

    public static int[] twoSum(int[] a, int target) {

        Map<Integer, Integer> seen = new HashMap<>();

        for (int i = 0; i < a.length; i++) {

            int need = target - a[i];

            if (seen.containsKey(need)) return new int[]{seen.get(need), i};

            seen.put(a[i], i);
        }

        return null;
    }
}

```

- Time: O(n), Space: O(n)

Time-window two-sum: keep last hour transactions in a queue + HashMap counts.

Problem 10: Multi-level cache (L1/L2/L3) + promotion + hit stats

The PDF specifies L1 LinkedHashMap, L2 map to SSD, and stats.

Design

- L1: LinkedHashMap access-order (LRU)
- L2: HashMap<id, path>

- L3: DB call (simulated)
- Promotion: if L2 hit, promote to L1

```

import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

public class MultiLevelVideoCache {

    static class VideoData {
        final String videoid;
        String payload; // simulate video metadata/content
        long version; // to simulate updates/invalidation

        VideoData(String videoid, String payload, long version) {
            this.videoid = videoid;
            this.payload = payload;
            this.version = version;
        }

        @Override
        public String toString() {
            return "VideoData{id=" + videoid + ", version=" + version + ", payload=" + payload + "}";
        }
    }

    // L2 entry: points to SSD path + keeps version (to detect stale)

    static class L2Entry {
        final String filePath;
        long version;

        L2Entry(String filePath, long version) {
            this.filePath = filePath;
            this.version = version;
        }
    }
}

```

```

    }

}

// ---- L1: LinkedHashMap for LRU (access-order) ----

static class L1Cache extends LinkedHashMap<String, VideoData> {

    private final int capacity;

    L1Cache(int capacity) {
        super(16, 0.75f, true); // access-order LRU
        this.capacity = capacity;
    }

    @Override

    protected boolean removeEldestEntry(Map.Entry<String, VideoData> eldest) {
        return size() > capacity;
    }
}

// ---- L2: SSD-backed simulation (LRU via LinkedHashMap for keys, plus path map) ----

static class L2Cache extends LinkedHashMap<String, L2Entry> {

    private final int capacity;

    L2Cache(int capacity) {
        super(16, 0.75f, true);
        this.capacity = capacity;
    }

    @Override

    protected boolean removeEldestEntry(Map.Entry<String, L2Entry> eldest) {
        return size() > capacity;
    }
}

```

```
}

// ---- L3: Database simulation ----

static class Database {

    private final Map<String, VideoData> store = new ConcurrentHashMap<>();

    public void put(VideoData v) {

        store.put(v.videoid, v);
    }

    public VideoData get(String videoid) {

        // Simulate slow DB call

        sleepMs(150);

        VideoData v = store.get(videoid);

        if (v == null) return null;

        // return a "copy" to simulate separate layer

        return new VideoData(v.videoid, v.payload, v.version);
    }

    public void update(String videoid, String newPayload) {

        VideoData v = store.get(videoid);

        if (v != null) {

            v.version++;

            v.payload = newPayload;
        }
    }
}

// ---- Stats ----

static class Stats {

    long l1Hits = 0, l1Miss = 0;
```

```

long l2Hits = 0, l2Miss = 0;

long l3Hits = 0, l3Miss = 0;

long l1TimeMs = 0;

long l2TimeMs = 0;

long l3TimeMs = 0;

void recordL1(boolean hit, long ms) { if (hit) l1Hits++; else l1Miss++; l1TimeMs += ms; }

void recordL2(boolean hit, long ms) { if (hit) l2Hits++; else l2Miss++; l2TimeMs += ms; }

void recordL3(boolean hit, long ms) { if (hit) l3Hits++; else l3Miss++; l3TimeMs += ms; }

String report() {

    long l1Total = l1Hits + l1Miss;

    long l2Total = l2Hits + l2Miss;

    long l3Total = l3Hits + l3Miss;

    double l1HitRate = l1Total == 0 ? 0 : (l1Hits * 100.0 / l1Total);

    double l2HitRate = l2Total == 0 ? 0 : (l2Hits * 100.0 / l2Total);

    double l3HitRate = l3Total == 0 ? 0 : (l3Hits * 100.0 / l3Total);

    double l1Avg = l1Total == 0 ? 0 : (l1TimeMs * 1.0 / l1Total);

    double l2Avg = l2Total == 0 ? 0 : (l2TimeMs * 1.0 / l2Total);

    double l3Avg = l3Total == 0 ? 0 : (l3TimeMs * 1.0 / l3Total);

    long totalReq = l1Total; // every request goes through L1 check

    double overallHit = totalReq == 0 ? 0 : ((l1Hits + l2Hits + l3Hits) * 100.0 / totalReq);

    double overallAvg = totalReq == 0 ? 0 : ((l1TimeMs + l2TimeMs + l3TimeMs) * 1.0 /
totalReq);

    return String.format(
        "L1: Hit Rate %.2f%%, Avg Time: %.2fms%n" +
        "L2: Hit Rate %.2f%%, Avg Time: %.2fms%n" +
        "L3: Hit Rate %.2f%%, Avg Time: %.2fms%n" +

```

```

        "Overall: Hit Rate %.2f%%, Avg Time: %.2fms",
        l1HitRate, l1Avg,
        l2HitRate, l2Avg,
        l3HitRate, l3Avg,
        overallHit, overallAvg
    );
}

}

// ---- Main cache system ----

private final L1Cache l1;           // videoid -> VideoData (fast)
private final L2Cache l2;           // videoid -> filePath/version (SSD)
private final Database db;          // slow store
private final Stats stats = new Stats();

// Track access counts (for promotion decision) :contentReference[oaicite:6]{index=6}
private final Map<String, Integer> accessCount = new HashMap<>();
private final int promoteThreshold;

public MultiLevelVideoCache(int l1Cap, int l2Cap, int promoteThreshold, Database db) {
    this.l1 = new L1Cache(l1Cap);
    this.l2 = new L2Cache(l2Cap);
    this.promoteThreshold = promoteThreshold;
    this.db = db;
}

// Public API: getVideo(videoid)

public synchronized VideoData getVideo(String videoid) {
    // ---- L1 lookup (simulate 0.5ms) ----
    long t1 = System.currentTimeMillis();

```

```

VideoData v1 = l1.get(videoid);
sleepMs(1); // approximate 0.5ms as 1ms (Java timers are coarse)
long l1Ms = System.currentTimeMillis() - t1;
if (v1 != null) {
    stats.recordL1(true, l1Ms);
    incrementAccess(videoid);
    return v1;
}
stats.recordL1(false, l1Ms);

// ---- L2 lookup (simulate 5ms) ----

long t2 = System.currentTimeMillis();
L2Entry e2 = l2.get(videoid);
if (e2 != null) sleepMs(5);

long l2Ms = System.currentTimeMillis() - t2;
if (e2 != null) {
    stats.recordL2(true, l2Ms);
    VideoData vFromDb = db.get(videoid); // in real: read from file; here we reuse db but
you can replace

    // That db call is slow; so instead simulate SSD read:

    // We'll fake SSD read:
    vFromDb = new VideoData(videoid, "SSD_CONTENT(" + e2.filePath + ")", e2.version);

    // Promote if access count > threshold :contentReference[oaicite:7]{index=7}

    int cnt = incrementAccess(videoid);
    if (cnt >= promoteThreshold) {
        l1.put(videoid, vFromDb);
    }
}

```

```

    return vFromDb;
}

stats.recordL2(false, l2Ms);

// ---- L3 DB lookup (simulate 150ms) ----

long t3 = System.currentTimeMillis();

VideoData v3 = db.get(videoid);

long l3Ms = System.currentTimeMillis() - t3;

if (v3 != null) {

    stats.recordL3(true, l3Ms);

    // Add to L2 first (frequently accessed) :contentReference[oaicite:8]{index=8}

    // Store a "file path" to represent SSD location

    String path = "/ssd/cache/" + videoid + ".bin";

    l2.put(videoid, new L2Entry(path, v3.version));

    incrementAccess(videoid);

    return v3;

}

stats.recordL3(false, l3Ms);

return null;

}

// Invalidate on content update (remove from L1 and L2)
:contentReference[oaicite:9]{index=9}

public synchronized void invalidate(String videoid) {

    l1.remove(videoid);

    l2.remove(videoid);

    accessCount.remove(videoid);

}

```

```
public synchronized String getStatistics() {
    return stats.report();
}

// ---- helpers ----

private int incrementAccess(String videoid) {
    int cnt = accessCount.getOrDefault(videoid, 0) + 1;
    accessCount.put(videoid, cnt);
    return cnt;
}

private static void sleepMs(long ms) {
    try { Thread.sleep(ms); } catch (InterruptedException ignored) {}
}

// ---- Demo main (matches sample I/O style) ----

public static void main(String[] args) {
    Database db = new Database();
    db.put(new VideoData("video_123", "DB_CONTENT_123", 1));
    db.put(new VideoData("video_999", "DB_CONTENT_999", 1));
    MultiLevelVideoCache cache = new MultiLevelVideoCache(
        10_000, // L1 capacity :contentReference[oaicite:10]{index=10}
        100_000, // L2 capacity :contentReference[oaicite:11]{index=11}
    );
    System.out.println("getVideo(\"video_123\")");
    System.out.println(cache.getVideo("video_123")); // likely L3 first time
    System.out.println();
    System.out.println("getVideo(\"video_123\") [second request]");
}
```

```
System.out.println(cache.getVideo("video_123")); // likely L2 hit then maybe promote
System.out.println();
System.out.println("getVideo(\"video_123\") [third request]");
System.out.println(cache.getVideo("video_123")); // likely L1 hit now
System.out.println();
System.out.println("getVideo(\"video_999\")");
System.out.println(cache.getVideo("video_999"));
System.out.println();
System.out.println("getStatistics()");
System.out.println(cache.getStatistics());
db.update("video_123", "DB_CONTENT_123_UPDATED");
cache.invalidate("video_123");
System.out.println("\nAfter update + invalidate, getVideo(\"video_123\")");
System.out.println(cache.getVideo("video_123"));

}

}
```